Optimizing Vector Magnitude on the Intel® Pentium® III Processor

By introducing the Intel streaming SIMD extensions instructions, the Intel® Pentium® III processor enabled game applications to operate on single precision floating-point data in SIMD (Single Instruction Multiple Data) fashion. This allowed most game developers using vector math to realize a significant increase in their application's speed by optimizing routines to take advantage of streaming SIMD extensions capabilities. The following application note shows how you can achieve the maximum speed increase in calculating vector magnitude when using streaming SIMD extensions.

Computing the Vector Magnitude

Figure 1 shows the C implementation of computing the magnitude of a vector. Consider a vector, **v** = (x, y). Using the Pythagorean theorem, we can determine the displacement of point **v** from the origin. The graphical representation of point **v** in two-dimensional space is shown in Figure 2.

```
typedef struct{
        float x;
        float y;
        float z;
        float pad;
   }vector;

float VectorMagnitude(vector *Vec)
{
     return (float)sqrt(Vec->x*Vec->x + Vec->y*Vec->y + Vec->z*Vec->z);
}
```

Figure 1. Vector magnitude in C

The distance of point **v** from the origin **o** is **v** = $\sqrt{(x^2 + y^2)}$

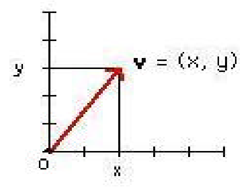

Figure 2

If v=(x, y, z) represents the displacement of point **v** in a three dimensional space from the origin as shown in Figure 3, then the magnitude of the vector **v** or the displacement of the point **v** from the origin is

$$v = \sqrt{(x^2 + y^2 + z^2)}$$

The above equation follows from the three dimensional generalization of the Pythagorean theorem.

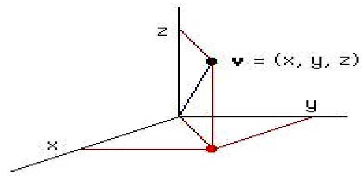

Figure 3

Optimizing the Vector Magnitude

Optimization for the vector magnitude function is done using *sqrtss* (square root), *rsqrtss* (reverse square root) *rcpss* (inverse) and *rcpss* and *rsqrtss* instructions in combination with Newton-Raphson method. The *sqrtss* instruction returns a result that is accurate to 24 bits for single precision numbers. While this instruction is simple, better performance can be achieved through approximations using other streaming SIMD extensions instructions. The *rcpss* and *rsqrtss* instructions are fast approximations that use an on-chip look up table to calculate the approximate reciprocal and square root reciprocal of a floating-point number. Though these approximations improve performance, they compare with less precision to *sqrtss* or x87 square root instruction. The *rcpss* and *rsqrtss* instruction returns a result accurate in the 12 most significant bits of the mantissa. Combined with Newton–Raphson method, the *rcpss* and *rsqrtss* instructions return a result accurate to 23 of 24 bits for single precision numbers. This approximation method offers a sizable performance gain compared to both the x87 square root and streaming SIMD extensions *sqrtss* instructions.

The above methods will be discussed in more detail in subsequent paragraphs. All optimizations in this application note were compiled using the Intel® C/C++ compiler version 4.5—the performance is compared with that of the C code shown in Figure 1. The processors' time stamp counters took the timings; for best results, call the function 50 or more times between the timing calls.

Optimization of the C code using *sqrtss* instruction is done as follows:

1. The vector is stored in a register and then multiplied by itself. Only one multiplication for all three elements (x, y, z) of the vector is required, compared to three multiplications in the C code shown in Figure 1.

2. The result is then shuffled so that each element is stored in a separate register.

3. The content of all the three registers are added to get the sum.

4. The square root of the sum is taken using *sqrtss* instruction.

The resulting performance is 16 percent better than the C version shown in Figure 1 on the Intel Pentium III processor. These results are accurate to 24 bits for single precision numbers. The optimized code is shown in Figure 4.

```
inline float Intrinsic_VecMag2_aligned(vector *Vec)
{
        __m128 vec1, vec2, vec3, vec4, vec5;
        float buffer [4];
        vec1 =_mm_load_ps((float *)Vec);
        vec2 = _mm_mul_ps(vec1, vec1);
        vec3 = _mm_shuffle_ps(vec2, vec2, 0x99);
        vec4 = _mm_shuffle_ps(vec2, vec2, 0xEE);
        vec3 =_mm_add_ss(vec3, vec4);
        vec5 = _mm_add_ss(vec2, vec3);
        vec5 = _mm_sqrt_ss(vec5)
        _mm_store_ss(buffer,vec5);
        return(buffer[0]);
}
```

Figure 4. Vector magnitude optimized in SIMD using *sqrt*

Even better performance can be achieved by using the *rsqrtss* and *rcpss* features of streaming SIMD extensions. With *rcpss* and *rsqrtss*, the vector magnitude function performed 50 percent faster than the C version shown in Figure 1 on the Intel Pentium III processor. However, the result using this method is only accurate to 12 bits. The code segment in Figure 5 implements *rcpss* and *rsqrtss* instructions.

```
inline float Intrinsic_VecMag3_unaligned(vector *Vec)
{
        __m128 vec1, vec2, vec3, vec4, vec5;
        float buffer[4];
        vec1 =_mm_load_ps((float *)Vec);
        vec2 = _mm_mul_ps(vec1, vec1);
        vec3 = _mm_shuffle_ps(vec2, vec2, 0x99);
        vec4 = _mm_shuffle_ps(vec2, vec2, 0xEE);
        vec3 =_mm_add_ss(vec3, vec4);
        vec5 = _mm_add_ss(vec2, vec3);
        vec1 = _mm_rsqrt_ss(vec5);
        vec1 = _mm_rcp_ss(vec1);
        _mm_store_ss(buffer,vec1);
        return(buffer[0]);
}
```

Figure 5. Vector magnitude in SIMD using *rcpss* and *rsqrtss*

The accuracy of the results from the *rcpss* and *rsqrtss* can be improved by using them in combination with Newton-Raphson method. You can learn more about the formula for the Newton–Raphson method in the paper "Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton–Raphson Method."

```
inline float Intrinsic_VecMag4_NewtonRaphson(vector *Vec)
{
        __m128 vec1, vec2, vec3, vec4, vec5;
        float buffer[4];
        float c[2] = {0.0f,0.0f};
        float *p = c;
        vec1 =_mm_load_ps((float *)Vec);
        vec2 = _mm_mul_ps(vec1, vec1);
        vec3 = _mm_shuffle_ps(vec2, vec2, 0x99);
        vec4 = _mm_shuffle_ps(vec2, vec2, 0xEE);
        vec3 = _mm_add_ss(vec3, vec4);
        vec5 = _mm_add_ss(vec2, vec3);
        vec1 = _mm_rsqrt_ss(vec5);
        vec1 = _mm_rcp_ss(vec1);
0       _mm_store_ss(p,vec1);
        vec2 = _mm_load_ss((float *)p);
        vec3 = _mm_load_ss(p);
        vec3 = _mm_rsqrt_ss(vec3);
        vec3 = _mm_rcp_ss(vec3);
         vec5 = _mm_set1_ps(1.934315383433);
        vec3 = _mm_sub_ss(vec3,vec5);
        vec2 = _mm_rsqrt_ss(vec2);
        vec4 = _mm_set1_ps(0.5);
        vec2 = _mm_mul_ss(vec2, vec4);
        vec2 = _mm_rcp_ss(vec2);
        vec2 = _mm_mul_ss(vec2,vec3);
        vec1 = _mm_sub_ss(vec1,vec2);
        _mm_store_ss(buffer,vec1);
        return(buffer[0]);
}
```

Figure 6 Vector magnitude using Newton-Raphson method

The code in Figure 6 implements *rcpss*, *rsqrtss* and the Newton–Raphson method. The vector magnitude function implemented using the Newton–Raphson method performs 40 percent faster than the C version on the Intel Pentium III processor and is accurate to 23 bits. The performance ratios mentioned for the implementation discussed throughout this application note are shown in Figure 7.

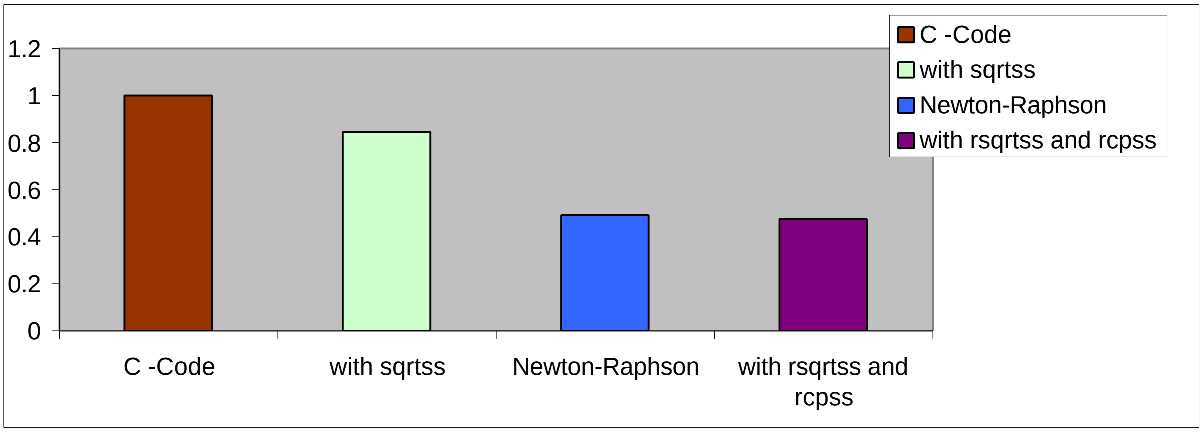

Figure 7. Benefits from *rcpss* and *rsqrts*

In conclusion, this application note has shown you different methods to increase speed in your applications when calculating vector magnitude and optimizing routines to take advantage of streaming SIMD extensions capabilities. As illustrated in Figure 7, the best option for developers to realize a significant speedup with an accuracy rate up to 23 bits in their game applications is by using the *rcpss* and *rsqrtss* in conjunction with Newton-Raphson.